Survivor: An Approach for Adding Dependability to Legacy Workflow Systems

Jean-Denis Grèze jg253@cs.columbia.edu
Gail E. Kaiser kaiser@cs.columbia.edu
Gaurav S. Kc gskc@cs.columbia.edu

Columbia University
Department of Computer Science
New York, NY 10027, United States
+1 212 939 7000

Abstract

Although they often provide critical services, most workflow systems are not dependable. There has been much research on dependable/survivable distributed systems; most is concerned with developing new architectures, not adapting pre-existing ones. Additionally, the research has focused on hardening, security-based defense, as opposed to recovery. For deployed systems, it is often infeasible to completely replace existing infrastructure; what is needed are ways to adapt existing distributed systems to offer better dependability. In this paper, we outline a general architecture that can easily be retrofitted to legacy workflow systems in order to improve dependability and fault tolerance. We do this by monitoring enactment and replicating partial workflow states as tools for detection, analysis and recovery. We discuss some policies that can guide these mechanisms. Finally, we describe and evaluate our implementation, Survivor, which modified an existing workflow system provided by the Naval Research Lab.

Keywords: System Fault Tolerance, CheckPointing, Workflow, Survivability

1 Introduction and Motivation

Dependability, fault-tolerance, survivability, etc. are rarely considered in workflow management system design. However, workflow systems increasingly provide enterprise or mission critical services, and thus should address dependability issues. Exception handling and “compensating” tasks (semantic undo) are established workflow mechanisms; however, these require the workflow author to carefully preconsider possible failures. In contrast, we seek to achieve some degree of dependability even without any effort on the part of the workflow author (although we can do better if any such special-case information is available).

Dependability is defined as the "property of a computer system such that reliance can justifiably be

placed on the service it delivers." This covers availability, reliability, safety, confidentiality, integrity and maintainability issues [Barbacci 1995]. We focus on providing availability, reliability, and maintainability through support for fault-detection, fault-analysis and fault-recovery.

Workflow Controllers are an interesting special case of distributed information systems to which it is logical to add dependability features. Because workflow processes are well-defined, they include machine-readable information that can be helpful in providing dependability. In particular, process definitions often highlight where branches may split, join, and synchronize, and how or where they interact with non-workflow components, along with specifics about the effects of individual tasks/computations. Ideally, a workflow process definition provides a precise model of that distributed computation. Workflow also provides the advantage that existing processes can be made more dependable by upgrading only the Workflow Controller, without any changes to the process definitions themselves. If the Controller supports mechanisms for fault tolerance, e.g., then all process definitions executed by that controller will be more dependable.

The long-term goal of our research is to not only add dependability to workflow, but to enable *survivability*. [Knight 2000] gives the following high level definition: "Informally by a survivable system we mean a system that has the ability to continue to provide service (possibly degraded or different) in a given operating environment when various events cause major damage to the system or its operating environment. Service quality, i.e., exactly what a system should do when damaged, seems to be something that should be determined by simple guidelines." Likewise, [Ellison 1999] refers to survivability as "the ability of a network computing system to provide essential services in the presence of attacks and failures, and recover full services in a timely manner." Survivable systems need to be engineered in a manner different than non-survivable systems [Knight 2000].

Some existing workflow implementations provide notations for describing anticipated errors within the process description, as well as mechanisms for handling these errors by the workflow management system (henceforth referred to as WfMS, see [Hollingsworth 1995]) such as Little-JIL [Wise 2000]. However, it is rarely possible for all potential failures to be described this way [Ellison 1999]. Trying to

account for a large number of distinct fault cases may make an otherwise simple workflow cumbersomely complex [Eder 1996]. The possibly prohibitive cost of the analysis required by the workflow author, and the risk that said analysis may not be exhaustive in considering all possible faults in the surrounding execution environment, are a strong motivation for developing automatic recovery mechanisms that work in a majority of situations. Figure 1 illustrates the type of recovery that such automation should provide.

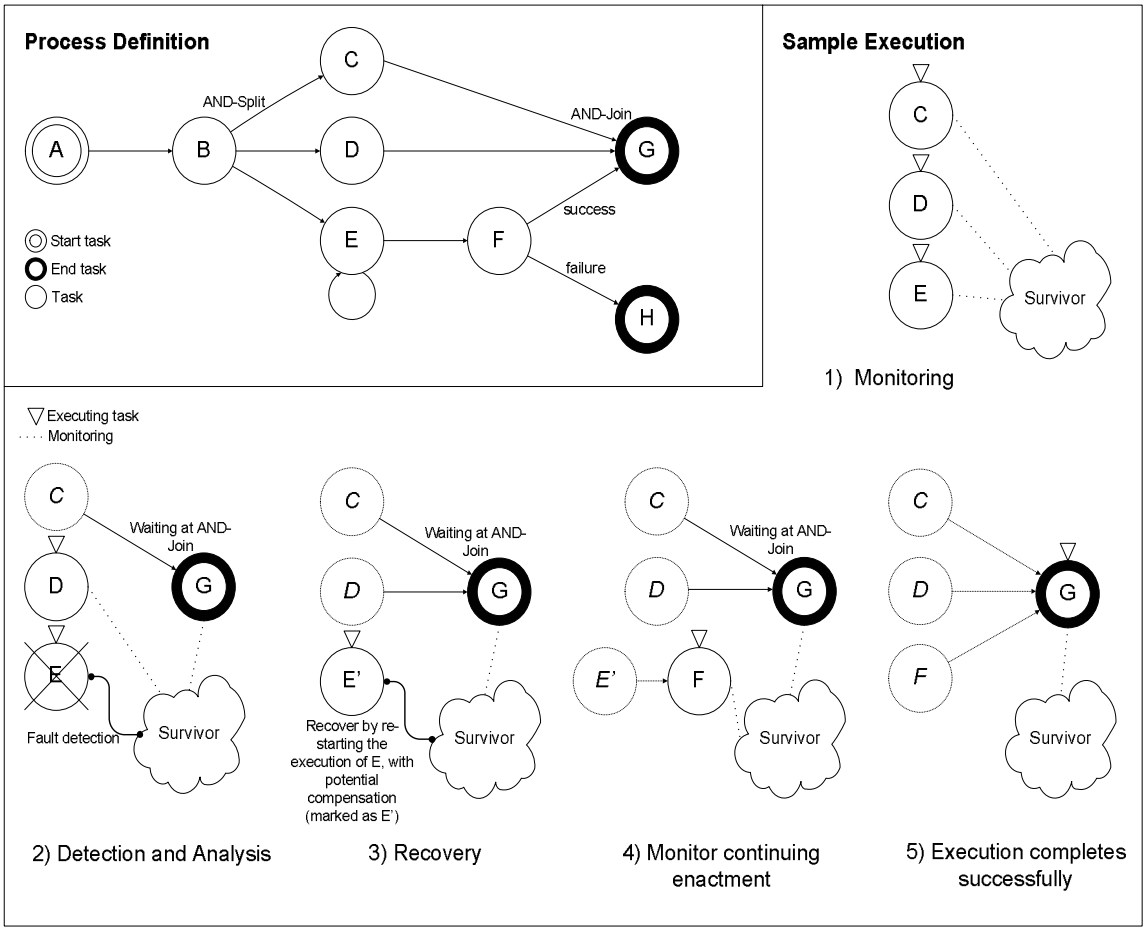

Figure 1

There has been much research on the dependability of distributed systems (e.g. [DSN 2000-2002]). However, most of it concerns the development of new architectures, not the adaptation of pre-existing systems. Our approach focuses on adding mechanisms to improve dependability in legacy workflow systems as opposed to designing and building from scratch. Since dependability criteria tend to change over time, we have focused on a flexible framework for implementation of a variety of dependability constraints and policies. Our initial set of policies, presented in this paper, focuses on recovery aspects as opposed to fault prevention.

We assume that the workflows of interest are enacted asynchronously in a decentralized manner, and that individual tasks generally have no knowledge of other tasks in the environment (with the exception of specifically synchronized tasks). Centralized control suffers from reliability and scalability problems [Kamath 1998a,b]. The decentralized model is characteristic of large unbounded computing environments (such as the Internet) and reflects coming trends in distributed computing [Ellison 1999].

In Section 2, we discuss existing approaches to providing fault tolerance and dependability to distributed workflow systems. In Section 3, we outline our model and the type of dependability policies that may be implemented using it. Section 4 describes our implementation of a prototype, which we call ***Survivor***, and its integration with a workflow system developed by the Naval Research Lab [Kang 1999]. In section 5, we analyze our results and draw conclusions. Finally, we discuss our ongoing and future work on fault tolerance, dependability and survivability.

2 Related Work

Many recent WfMS's support distributed enactment and control, and over the last few years some have been concerned with dependability and correctness. [Eder 1996] describes ways in which recovery may be ensured by integrating transactional features into a WfMS. It differentiates between document-oriented and process-oriented workflow and analyzes the failures and recovery that are relevant to both. Specifically, it details how operations such as undo, backward recovery, restart, etc. may be used as recovery mechanisms. [Casati 1998] provides analysis of the different types of exceptions that may affect a workflow enactment and presents ways to handle them as used on their WIDE workflow project. [Klein 2000] discusses a knowledge-based approach to exception processing and recovery at enactment time. "Normal" process models are analyzed so that exceptions may be better detected, even anticipated, when "non-normal" situations occur; pre-defined avoidance/resolution handler are then invoked.

The Little-JIL project [Wise 2000] provides a workflow language for programming coordinated agents. It focuses on the specification of control structures, and less so on the agents for executing the work. Little-JIL provides a powerful exception handling mechanism for recovery from failures. However, their mechanisms are meant to handle mostly pre-defined local exceptions and provide little aid in

recovering from unanticipated and/or non-local failures.

METEOR [Cardoso 2001] implements a survivable architecture for WfMS's, where dynamically changing the workflow process definition is taken as the approach to adaptation and evolution. Additionally, the same group at the U. of Georgia has done work on a variety of METEOR extensions such as the reuse of exception handling experiences when confronted with new cases [Luo 2000], WebWork, a distributed workflow system using Web technology [Miller 1999], and cross-organizational conflict resolution [Luo 2003].

[Knight 2001]'s Willow workflow architecture, developed jointly by the University of Virginia, the University of Colorado at Boulder and the University of California at Davis, implements an approach to survivability by combining fault avoidance, elimination and tolerance. Avoidance is marked by disabling vulnerable network elements in case of threat; elimination replaces faulty software elements while tolerance works by reconfiguring the system (called "posturing")., which may involve changes in functionality). Survivability is ensured by a control structure that is separate from the workflow itself which may also be used for "deploying functionality enhancements". Though Willow's reactive controller was an inspiration for the design of our observer component, our system differs because of its focus on asynchronous automatic recovery. Willow also requires in-depth knowledge (of both the application and faults), and active control over the underlying application to provide survivability. Furthermore, their system is meant to operate on a separate network with specific computational and communication needs.

3 Model and Architecture

Our model consists of four main elements: monitoring, replication, versioning (see 3.2 and Appendix) and observation. We describe each, explain how they work together to accomplish recovery and discuss dependability policies that may be supported.

3.1 Monitoring

Our model calls for an evolving set of nodes called Monitors, which are determined dynamically at runtime. Before a particular task begins executing on a particular node, a set of Monitors is elected from nodes in the workflow (i.e. processors that may execute tasks) to observe that task's execution. Once the

task is completed, the set verifies that control is transferred to the next task/node(s), at which point another set of Monitors is selected to oversee each follow-up task. Upon election of this set(s) of Monitors, the first set resigns. In addition to monitoring an individual task's execution and handoff to the following task(s), Monitors observe each other to enforce their own dependability via redundancy.

Monitors may implement a variety of detection algorithms (for faults, partitions, intrusions, etc.), e.g., [Gärtner 1999] [Lee 1998]; in our implementation we only provide detection of local exceptions, node failures and a mechanism for detecting network-failure- and partition-caused errors after the fact. The specific details of detection are not our main concern. When one or more Monitors fail (or become unreachable, compromised, etc.), the remainders recruit replacements. In the case of a fault, the Monitors elect a leader from among themselves (there are algorithms to do this, e.g. [Abu-Amara 1988] [Lynch 1996], as well as criteria for electing good leaders, e.g. [Singh 1994]), and begin analysis and recovery.

There are many policies for selecting the set of Monitors to oversee a given task. It is generally preferable to pick nodes that are unlikely to fail or be removed from the workflow enactment system; there has been literature on algorithms for similar selections, e.g. [Subhlok 1999]. It may be more efficient to keep the same set of monitors for consecutive tasks as opposed to re-electing every step of the way as stated above. Some policies might mandate more Monitors for some tasks than others or otherwise place special additional requirements on which nodes might serve as Monitors. As part of Monitors' own fault tolerance, they may choose to add or remove nodes from their set during execution in order to tradeoff maximizing dependability vs. minimizing resource usage.

3.2 Replication

In the worst case, the WfMS is capable of recovering only by restarting the workflow from the beginning. A factor of dependability is the ability to provide services in a timely manner, so when restarting workflow execution it is paramount to do so as near to the failure as possible and to repeat the minimal amount of work. Our approach provides a facility to restart a workflow at various intermediate points, ideally any previously executed task. In order to do so, however, it is required to have saved partial executions along the way. We employ Replicators to do this. Just as Monitors are selected to

oversee task execution, Replicators are dynamically selected to store partial executions of the workflow. Following a failure, it may then be possible to restart a workflow midway through its enactment by locating an appropriate partial execution for the failed branch.

After a task's execution, control, state and application data is stored as a Version on a set of remote Replicators specific to that task. Unlike Monitors, which require the ability to oversee execution, Replicators need the ability to store Versions as long as the workflow is active. They are called Versions because the data that the workflow as a whole manipulates may be seen to change throughout the computations; Versions are snapshots of this information.

An issue with replication is potentially huge storage costs. Since we are storing the results of each task multiple times, and since a workflow is generally made up of many tasks, there is potential for exceedingly large resource usage. In addition to storage costs, we must consider the bandwidth needed to transport Versions to the different Replicators. A few schemes may be used to alleviate this concern. One is to expire Versions slowly over time, so that there are fewer replicated copies of older Versions than of more recent ones. At critical points, the system may choose to mark certain Versions as non-expiring and save to stable storage for a permanent history. Also, because snapshots are likely to be similar, we anticipate the use of established version storage algorithms, such as “diff”s to previous or next Versions, as helpful in keeping down storage costs [Hunt 1998].

Unlike Monitors which perform mutual checks to ensure fault tolerance and recruit replacements as necessary, Replicators do not; once a Version has been replicated, there is currently no mechanism to ensure that copies of it will always be available. The rationale behind this design decision is that we can restart a task from a slightly older Version(s) should the latest Version be unavailable. While this may require more re-computation, we view this as an acceptable tradeoff between computation and storage.

When the Monitor leader begins recovery, it tries to find the most recent Version for its in-progress workflow branch. When a new set of Monitors is selected for an initializing task, those Monitors are informed of the selected Replicators for some number of recent previous tasks. This list is implemented as a deque of constant size, and can be used to find a Replicator following a fault. The first operating

replicator found would have the latest available version from which to restart execution. If all the Replicators in the deque are unavailable, e.g., due to faults or network partitions, the Monitor leader must then search the reachable nodes to find an acceptably recent available Version. (See Appendix)

3.2.1 Separation of Replication and Monitoring

An important component of our design is that monitoring and replication are separate. We justify this as they have different purposes and a different impact on resources. Monitoring should be relatively lightweight, although processing, bandwidth, and memory requirements will depend on the detection model(s) employed. Replication, however, is heavyweight, usually with considerably more storage and bandwidth usage than monitoring. If all Replicators for the last version fail, we may always fall back on an older Version to restart from. If all the Monitors for a given task fail, and detection is left to the Monitors of a parallel branch at the next synchronization point, however, recovery may be very inefficient. Thus there should be a larger number of Monitors than Replicators for any given task.

3.3 Observation

Although we assume that the workflow control is decentralized, recovery can benefit from a "big picture" of the workflow enactment, which would necessarily be centralized in some sense, but possibly replicated. This would be helpful for validating a workflow's execution or enacting sophisticated detection and recovery policies that cannot be implemented locally [Gärtner 1999], e.g. diagnosing deviations from expected general behavior as part of intrusion tolerance [Stavridou 2001]. The model includes mechanisms for both independent and coordinated recoveries as needed.

Our observation mechanism, the SENS (Start-End-Node-Set), is a set of nodes, each of which keeps a copy of the workflow-wide state. Before and after each task execution, the Monitors are responsible for informing the SENS of the state of their workflow branch. As such, the SENS always has a fairly accurate picture of the workflow as a whole, but has little impact on efficiency as it is updated asynchronously.

In principle, the SENS might also store workflow-wide control states that could be utilized by the Monitors during recovery. For example, the SENS could keep track of how often a particular branch has had faults, or of how many times a particular node has failed. This information should be helpful in

detecting recurring patterns and selecting alternative recovery strategies. The SENS' "big picture" might also be needed in detecting threats whose patterns cannot be identified locally, as well as recovering from non-local faults (those that affect multiple branches at once) [Knight 2001]. However, at present we have implemented only the simpler, passive SENS, which receives task notifications but does not yet analyze or act upon this information.

Because the SENS is essentially a centralized component, one may be concerned about its effect on scalability and its lack of reliability [Kamath 1998a,b]. For this reason one instance of the SENS is responsible only for the execution of one workflow, even if the WfMS is simultaneously enacting multiple workflows. Additionally, policies might determine how often Monitors should report to their SENS. Reliability of the SENS itself is achieved by implementing it as a set of nodes, as opposed to a single node. The SENS is selected before the workflow starts executing, so all subsequent Monitors know which nodes to report to. Consider the worst-case possibility where all the nodes in a SENS fail over time. One non-solution is simply to stop reporting, since the SENS is optional anyway, but then one loses its benefits. Another is to ensure that the SENS is initially large enough so that this scenario will rarely occur. A more heavyweight approach is to have the SENS recruit new members as needed, and reply to Monitor reports with updates of its state. We discuss the SENS further in Section 5.2.

3.4 Recovery: How it all works together

Imagine an ongoing workflow enactment with multiple concurrently executing branches, each with its own set of Monitors, and previous Versions of tasks have already been replicated. One of the nodes currently executing a task fails; this is immediately noticed by that task's Monitors, which proceed to elect a leader from among themselves to oversee analysis and recovery. At this point the Monitors switch from monitoring the now-failed task to overseeing the recovery process by the newly-elected leader. The first step is to inform the SENS that a task failed. The default recovery mechanism, invoked in this case, is to find a Version from which execution of the failed task may be attempted once again. As detailed in 3.2, an appropriate Version may be found by looking at the leader's (or another Monitor's) deque of previous Replicators; if all of these Replicators are unavailable, a search of neighboring nodes is required.

Assuming a Version from which to restart is found, the leader proceeds to find a suitable node for execution using the WfMS' task assignment mechanism. Once such a node has been found, a new set of Monitors is then elected to oversee execution, the leader informs the SENS that recovery has taken place, and then resigns along with the other original Monitors.

Note that if the enactment system has no available node to execute the task, the workflow may be deemed impossible to enact and execution aborted. This should not be considered an issue with our approach to dependability as there are environments where even degraded service may be impossible to provide. Finally, in case the original fault is due to a local exception, the node selected to restart execution may be the one on which the task initially failed (although another node should be selected in case of repeated failure). Figure 2 illustrates the recovery of a task which is itself dependent on multiple branches of the workflow enactment.

3.5 Dependability Policies

We consider two classes of policies, those that presume there are no changes to the original process definition, and those that can exploit additional failure-recovery information added to the process definition. Without changes to pre-existing process definitions, restarting the workflow from the most recent possible Version is an acceptable recovery scenario; the system benefits from leveraging heightened detection mechanisms implemented by Monitors to avoid restarting from scratch. Additionally, the Monitor leader may implement policies dealing with repeated failures, which might be due to an inappropriate task/node assignment model or an inherently faulty process definition, such as retrying using a different node selection scheme or restarting from an earlier point in the disrupted process (i.e., an earlier Version). This is by itself quite an improvement over non-dependable workflow.

For those process definitions that have been enhanced, however, more complex recovery scenarios can be considered, improving dependability. Consider a process where some task implementations have been upgraded to include more features; new failures might occur that weren't seen before the upgrade. However, if we include additional process data such as references to pre-upgrade implementations of the tasks, Monitors can switch to the older implementations at runtime in case of failures: it is often

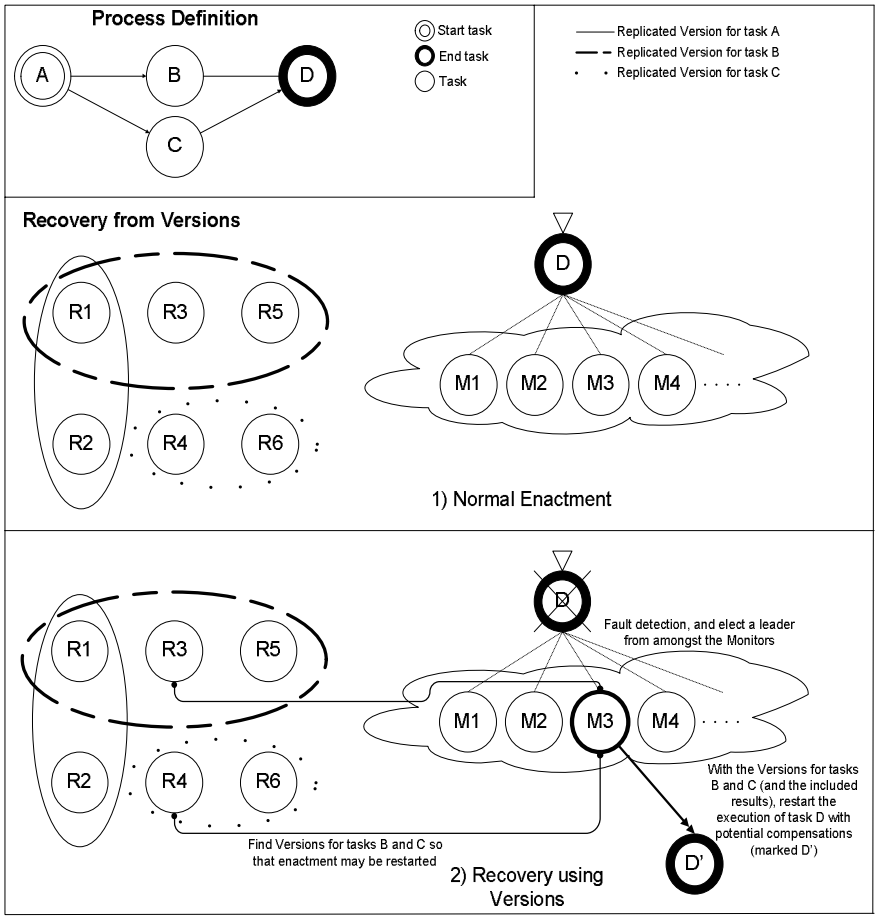

Figure 2

preferable to have working services even if they offer fewer capabilities. Due to the flexibility of our versioning mechanism, it is also possible to implement policies such as undo. If we stored only the results of the last task's enactment, there would be no way to restart execution from earlier points. However, we potentially provide access to replicated Versions of all previous tasks, and/or a series of "diffs" indicating changes to workflow data, with which we may be able to rollback multiple steps. Another interesting concept for recovery is that of compensation [Eder 1996], especially when rollback of certain side-effects is impossible. Monitors may then be responsible for more than simply restarting the initial workflow; with appropriate recovery information, they may choose to start explicitly compensating tasks or alternate workflows, as (possibly temporary) measures to mitigate risk, delays or unavailability [Kamath 1998a].

4 Survivor Implementation

Survivor is an extension to a 3rd-party legacy WfMS developed by the Naval Research Laboratory.

4.1 NRL Workflow

The Navy Research Lab's (henceforth NRL) workflow designer and runtime engine constitute a Multi-Level-Secure (henceforth MLS) implementation of workflow in the Java programming language. An issue with MLS-workflow is the impossibility of any single entity tracking the whole enactment due to security constraints [Kang 1999]. Specifically, there is often no way to tell whether tasks in other security domains have executed properly and successfully. The designer-component supports building process definitions spanning multiple security domains, and then splits such a workflow into separate security-domain-specific definitions. The runtime engine enacts these processes and provides mechanisms for safely transferring data and control between domains. NRL's workflow supports the usual branching, joins, etc. and is based on University of Georgia's METEOR [Cardoso 2001] [Kang 2001].

4.2 Survivor

4.2.1 Step 1: Implementing Survivor

Survivor is implemented in Java, and uses Worklets, our Java-based mobile agent technology [Valetto 2001] during monitoring to communicate with the executing task's node, during replication for transporting Versions to Replicators, and for transitioning to the successor task for recovery. The Worklets toolkit enabled us to implement the system more rapidly; our model can be used with other transport mechanisms if desired.

We implemented and tested Survivor (Monitors, Replicators, Versions, SENS observation) independently from NRL's workflow system, aiming for a workflow-implementation-independent design.

4.2.2 Step 2: Adapting the NRL System to Fulfill Survivor's Assumptions of the Underlying WfMS

NRL's workflow engine (without Survivor) does not support determining at runtime where tasks are to execute, but instead utilizes pre-assigned mappings, created in the designer, from the process definition to the execution environment. In order to implement our recovery mechanisms, we need dynamic task routing. We had developed a prototype dynamic routing system for the NRL system, extending components for workflow initiation, completion and control. This prototype was based on a bypass of the default RMI-based inter-node transport mechanism of the NRL system, coupled with the provision for multiple processing nodes for each task.

Improving upon the original idea of multiple processors for given tasks, we introduced the concepts of node capabilities and services, as well as task requirements. To enact a given task on a given node, its task requirements must be a subset of that node's capabilities. Our implementation provides nodes with only partial knowledge of resources available for enactment. At runtime, nodes look among their immediate neighbors for the capabilities for the next task; if no such node exists, the task migrates outward and repeats the process. Whenever a new node joins the enactment environment, it informs its neighbors of its capabilities. If the NRL system had included these facilities natively, we could have leveraged them (Survivor is independent of the specific task routing implementation, as long as it is dynamic). We also changed the inter-node transport mechanism of the system from Java RMI to Worklets to simplify integration.

4.2.3 Step 3: Integrating Survivor with the Modified NRL System

Finally, we had to connect NRL's engine to Survivor, while minimizing changes to NRL's codebase. We modified NRL's code to expose the scheduling of tasks, so Survivor could intercept transitions from one task to the next, but otherwise made no changes to their task execution mechanisms. The most significant challenge that arose during integration was attaining a good enough understanding of their workflow engine to make our changes without affecting its features. There was no documentation other than the source code itself, and the original developers had left NRL and were no longer available.

4.2.4 Step 4: Validation

We validated our approach by looking at the following four characteristics: proper enactment in a fault-free environment, dependability while introducing single local faults, dependability while introducing multiple faults, and dependability of Survivor itself.

Our initial goal was to confirm that non-modified process definitions would execute correctly on the Survivor-enabled workflow runtime. We tested capabilities and requirements and our implementation of dynamic routing. We tried boundary cases such as executing a process definition on an environment without the required node capabilities. As expected, the workflow waited until nodes with the needed capabilities joined the environment, and failed if such nodes did not materialize within a time bound.

Overall, processes executed successfully, individual tasks were monitored, and their data was replicated along the way. We also developed a small auxiliary application to display information from all monitors and replicators to track Survivor's progress.

The second step was testing proper recovery in case of local faults. We manually caused failures at some nodes during, before, and after execution. As expected, Monitors identified these faults and recovery was ensured by restarting the workflow from a suitable Version as per our basic policy.

Thirdly, we tested more complex scenarios involving multiple faults on multiple branches. For example, we forced two identical workflows to execute at the same time, which might occur as recoveries on both sides of a network partition. As expected, one of the two executions was eventually identified as a Version Clash (see Appendix), and the unwanted duplicate stopped.

The current implementation of Survivor is not completely successful with multiple parallel branch failures. Since OR-Splits are often eventually followed by Joins, there may be implicit dependencies between resulting branches. Unfortunately, our current implementation does not support a mechanism for handling implicit dependencies, as opposed to explicit dependencies defined in the workflow process definition itself, e.g. by a Join task. Our implementation does work successfully when failing branches have explicit (or no) dependencies.

Our fourth round of validation was aimed at Survivor components themselves. We forced faults on all Replicators of the last version, successfully forcing usage of an older previous Version for restarting the workflow. We also tested the dependability of Monitors by forcing failures. As expected, new Monitors were selected to compensate for the faults. In case all Monitors for a given task fail, execution of the process still continues as long as the task itself has not failed; after the current task finishes execution, a new set of Monitors is elected for the next task. Dependability of Monitoring is generally violated only for the currently executing task in the case of failure of a whole set of Monitors at once.

5 Evaluation

This section informally studies the ease of integrating Survivor with NRL's workflow engine, and then gives a brief qualitative analysis of our approach to fault-discovery and recovery.

5.1 Ease of Integration

As stated earlier, we designed Survivor to be transparent to the underlying workflow system. Although we have integrated Survivor with only one pre-existing workflow system to date, we feel that we can claim acceptable transparency. Data from our integration with NRL's workflow system shows that there is little coupling between Survivor and the underlying workflow implementation, and that the integration with Survivor required very few changes to the WfMS's inner workings.

Component	Lines of Code (LOC)	Files	Classes	Comment
NRL	4024	47	48	
Worklets	4007	20	64	
Survivor	3610	22	47	
Dynamic Routing	1112 new + 34 changed to NRL	11 files in 9 files	18	28.4% (LOC) extension to NRL
Survivor Integration	369 new + 35 changed to NRL	6 files	6	9.3% (LOC) extension to NRL

The size of NRL's runtime is approximately 4000 lines of code (LOC) in the Java programming language, spread across 47 files implementing 48 classes/interfaces. As discussed in Section 4.1, NRL's workflow engine previously had no support for dynamic routing of tasks. Hence, the first step was to add this facility. This preparation phase also included integration of Worklets for transport purposes, e.g., delivering task definitions to the newly routed tasks. We changed 34 LOC in 9 of NRL's source files, and we extended and added classes by 1112 LOC in 11 new files (18 classes). Adding dynamic routing and Worklets transport required a fair amount of work within NRL's engine. Though 34 lines of direct changes represent less than 1% of the existing NRL runtime codebase, 1112 lines of add-ons swelled the code by more than 28.4%. We had to become very familiar with their engine to add dynamic routing; this integration step could not be considered transparent. However, all this effort should not be required across-the-board for Survivor integration, as many other distributed workflow systems already provide mechanisms for dynamic routing and distributing tasks to the selected task processors.

Survivor itself was much more straightforward to integrate. On its own, Survivor stands at 3610 LOC (excluding its internal transport, which is also handled by Worklets) in 22 files implementing 47 classes. Integration of Survivor with the dynamic-routing version of NRL's runtime required changes to 35 LOC of NRL’s codebase, and 369 LOC in 6 files implementing 6 "Survivor-enabling" new classes. In

total, the “glue” code between Survivor and the enhanced NRL WfMS consists of roughly 400 lines of code. This adaptation involved extensions to 3 classes from NRL (out of 48), with the following understanding of their runtime required – just how to start a task execution and how to intercept results after completion of a task’s execution. Among the 369 “Survivor-enabling” LOC, 146 or approximately 40% was code “cut and paste”-ed from the original classes into the new extended classes. Overall, NRL's codebase was extended by 9.3% of its original size, but only 6.4% was significant.

Survivor itself was implemented by the first author; integration with NRL's workflow system and all Worklets-based communication was done by the third author. Survivor was coded and tested independently, and little knowledge of Survivor's inner workings was required to incorporate it into the enhanced NRL workflow runtime. The two developers initially collaborated on the NRL enhancements (dynamic routing), but then worked independently until integration testing. No changes to Survivor's design were needed in order to integrate it with the enhanced NRL's workflow.

5.2 Qualitative Analysis

As an architecture for implementing baseline dependability and survivability, our approach is successful. We have been able to implement simple policies such as restarting from the most recent available Version, to more complex recovery involving multiple tasks. We have not yet fully tested Survivor’s flexibility to handle more sophisticated policies, such as transactional rollback or containing intruders, which are concerns for future research.

Non-local fault recovery remains relatively weak due to the asynchronous nature of our implementation, since we do not yet exploit the SENS’ “big picture” to coordinate recoveries across branches. In the absence of more detailed process definitions, we are forced to restart dependent branches at their splitting task, which often requires redoing more work than if we had process semantics or more fine grained control available, e.g., a proactive SENS. However, as noted previously, we are concerned about the implications on scalability and reliability with a more centralized SENS; although it is redundantly replicated, it functions in a centralized manner. We are looking at solutions for this by providing runtime facilities for mobilizing “neighborhood watches” coordinating dependent threads of

execution and their recovery in case of failure.

Another limitation stems from typical workflows' dependencies on external data. We provide no mechanisms for undoing, rolling back or compensating changes to external data such as that contained in a database, since in many cases the volume may be enormous. We assume such components provide recovery mechanisms of their own, which Survivor can interact with to provide dependability. If not, the result may be that we enact some transactions multiple times as their corresponding tasks are re-executed as part of recovery. This is not acceptable: imagine a single withdrawal from an ATM resulting in multiple debits from the database keeping track of your account. Side effects caused by external mechanisms also hinder us in our goal to maintain correct execution of the workflow, such as an ATM which repeatedly outputs $20 bills.

Unfortunately, this means that Survivor can only guarantee correct recovery on internal workflow data; in general, Survivor does not know which tasks involve updating external data or other potentially problematic side-effects. This problem can be ameliorated by making it explicit in the process definitions which tasks affect, or are affected by, external components – and then include compensating tasks or other specialized recovery facilities in the process definition.

6 Future Work

We are currently exploring the feasibility of integrating Survivor with several other workflow systems, including Cougaar (http://www.cougaar.org), Little-JIL [Wise 2000], and BPEL4WS [Curbera 2002] (all of which we are using in other projects in our lab) as well as the possible adaptation of our approach to other forms of distributed computing. We would like more data on the flexibility of our approach in working with different WfMSs, particularly additional integrations performed by others than the main Survivor developer. In the longer term, we would like to take a deeper look at survivability, including semantics of "graceful degradation" anticipated for survivable workflow systems. Finally, we have recently initiated joint research with Prof. Sal Stolfo at Columbia University on application-level intrusion detection, which we hope to incorporate into our Monitoring.

6.1 Acknowledgements

We would like to thank Jim Tracy and Judith Froscher at NRL, and Myong Kang formerly of NRL, for providing us with their workflow system.This work was funded in part by the Office of Naval Research monitored by Naval Research Laboratory N00014011044. The Programming Systems Laboratory is also funded in part by Defense Advanced Research Project Agency under DARPA Order K503 monitored by Air Force Research Laboratory F30602-00-2-0611, by National Science Foundation grants CCR-02-03876, EIA-00-71954, and CCR-99-70790, and by Microsoft Research. In addition, we would like to extend a special thanks to Janak Parekh, Philip N. Gross, Suhit Gupta and Giuseppe Valetto.

7 References

[Abu-Amara 1988] H. Abu-Amara. Fault-Tolerant Distributed Algorithm for Election in Complete Networks. In *IEEE Transactions on Computers*, 37(4): 449-453, April 1988.

[Barbacci 1995] M. Barbacci, M.H. Klein, T.A. Longstaff, and C.B. Weinstock. Quality Attributes. *Technical Report*, CMU/SEI-95-TR-021, December 1995.

[Cardoso 2001] J. Cardoso, Z. Luo, J. Miller, A. Sheth, and K. Kochut. Survivability Architecture for Workflow Management Systems". In *Proceedings of the 39th Annual ACM Southeast Conference (ACMSE'01)*, pages. 207-216, Athens Georgia, March 2001.

[Casati 1998] Casati, Fabio. A Discussion on Approaches to Handling Exceptions in Workflows. *CSCW Workshop on Adaptive Workflow Systems (CSCW-98 Workshop)*, Seattle, WA, November 1998.

[Curbera 2002] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, S. Weerawarana. Business Process Execution Language for Web Services, Version 1.0. July 2002. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/

[DSN 2000-2002] Proceedings, *DSN 2000-2002*.

[Eder 1996] J. Eder, and W. Liebhart. Workflow Recovery. In *Proceedings of the First IFCIS International Conference on Cooperative Information Systems (CoopIS'96)*, pages 124-134, Brussels, Belgium, 1996.

[Ellison 1999] R.J. Ellison, D.A. Fisher, R.C. Linger, H.F. Lipson, T.A. Longstaff, and N.R. Mead. An Approach to Survivable Systems. At the *NATO IST Symposium on Protecting Information Systems in the 21st Century*, Washington, DC, October 1999.

[Gärtner 1999] F.C. Gärtner. Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments. In *ACM Computing Surveys (CSUR)*, 31(1):1-26, March 1999.

[Hollingsworth 1995] D. Hollingsworth. The Workflow Reference Model. From the *Workflow Management Coalition (WfMC)*, January 1995. http://www.wfmc.org/

[Hunt 1998] J.J. Hunt, K-P. Vo, and W.F. Tichy. Delta Algorithms: An Empirical Analysis. In *ACM Transactions on Software Engineering and Methodology*, 7(2): 192-214, April 1998.

[Kamath 1998a] M. Kamath, and K. Ramamritham. Failure Handling and Coordinated Execution of Concurrent Workflows. In *Proceedings of 14th International Conference on Data Engineering*, pages 334-341, Orlando, Florida, February 1998.

[Kamath 1998b] M. Kamath, and K. Ramamritham. Pragmatic Issues in Coordinated Execution and Failure Handling of Workflows in Distributed Workflow Control Architectures. *Technical Report*, 98-28, Dept. of Computer Science, Univ. of Massachusetts, August 1998.

[Kang 1999] M.H. Kang, J.N. Froscher, A.P. Sheth, K. Kochut, and J.A. Miller. A Multilevel Secure Workflow Management System. In *Proceedings of the 11th Conference on Advanced Information Systems Engineering*, pages 272-285, Heidelberg, Germany, 1999.

[Kang 2001] M.H. Kang. Private communication, January 2001.
[Klein 2000] M. Klein and C. Dellarocas. A Knowledge-Based Approach to Handling Exceptions in Workflow Systems. *Journal of Computer Supported Collaborative Work*, 9(3/4):399-412, August 2000.
[Knight 2000] J.C. Knight, and K.J. Sullivan. On the Definition of Survivability. *Technical Report*, CS-TR-33-00, Dept. of Computer Science, Univ. of Virginia, 2000.
[Knight 2001] J.C. Knight, D. Heimbigner, A.L. Wolf, A. Carzaniga, J.Hill, P. Devanbu, and M. Gertz. The Willow Architecture: Comprehensive Survivability for Large-Scale Distributed Applications. *Technical Report*, CU-CS-926-01, Dept. of Computer Science, Univ. of Colorado, December 2001.
[Lee 1998] W. Lee, and S.J. Stolfo. Data Mining Approaches for Intrusion Detection. In *Proceedings of the 7th USENIX Security Symposium (SECURITY'98)*, San Antonia, Texas, January 1998.
[Luo 2000] Z. Luo, A. Sheth, K. Kochut, and J. Miller. Exception Handling in Workflow Systems. In *Applied Intelligence: the International Journal of AI, Neural Networks, and Complex Problem-Solving Technologies*, 13(2):125-147, September/October 2000.
[Luo 2003] Z. Luo, A. Sheth, K. Kochut, and B. Arpinar. Exception Handling for Conflict Resolution in Cross-Organizational Workflows. To appear in *Distributed and Parallel Databases Journal*, Spring 2003.
[Lynch 1996] N.A. Lynch. Distributed Algorithms. Pages 25-46, 475-523, 1996.
[Miller 1999] J. Miller, A. Sheth, K. Kochut, and Z. Luo. Recovery Issues in Web-Based Workflow. In *Proceedings of the 12th International Conference on Computer Applications in Industry and Engineering (CAINE-99)*, pages 101-105, Atlanta, Georgia, November 1999.
[Singh 1994] S. Singh, and J.F. Kurose. Electing Good Leaders. *Journal of Parallel and Distributed Computing*, 21(2):184-201, May 1994.
[Subhlok 1999] J. Subhlok, P. Lieu, and B. Lowekamp. Automatic Node Selection for High Performance Applications on Networks. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'99)*, Atlanta, Georgia, May 1999.
[Stavridou 2001] V. Stavridou, B. Dutertre, R.A. Riemenschneider, H. Säýdi. Intrusion Tolerant Software Architectures. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX II'01)*, 2(2):1230-1241, June 2001.
[Valetto 2001] G. Valetto, G. Kaiser, and G. S. Kc. A Mobile Agent Approach to Process-based Dynamic Adaptation of Complex Software Systems. In *8th European Workshop on Software Process Technology*, pages 102-116, June 2001.
[Wise 2000] A. Wise, A.G. Cass, B.S. Lerner, E.K. McCall, L.J. Osterweil, and S.M. Sutton, Jr.. Using Little-JIL to Coordinate Agents in Software Engineering. In *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, pages 155-164, Grenoble, France, September 2000.

Appendix - Versions

Versions are identified/named by their workflow execution ID, a list of previously executed tasks, their corresponding nodes, and local start/end timestamps for the executions. For clarity, we use unique task names below; although this is not required as Survivor utilizes timestamps. In the first example,

WfID - T_A N_1 $TS_{A\,start}$ $TS_{A\,end}$ - T_B N_2 $TS_{B\,start}$ $TS_{B\,end}$ - T_C N_3 $TS_{C\,start}$ $TS_{C\,end}$ - …

workflow "Workflowid" consists of tasks T_A executed at N_1, T_B at N_2, T_C at N_3, and so on. The actual data stored corresponds to the result of the *last* named task with an end timestamp. (For brevity, we omit timestamps for the rest of this section.)

Any Version with a common root may be used to recover from a failed task. The Version with the largest common root is preferred as it will require less work to be redone. For example, if T_C is currently executing at N_3 and fails, any of the following Versions would allow us to restart execution:

WfID - (only contains data to start workflow)
WfID - T_A N_1 -
WfID - T_A N_1 - T_B N_2 - (most recent or best).

Additionally, we reserve the special task "Re" to identify recovery, along with a timestamp for detection of the fault (start) and for the completion of the recovery process (end). For example if recovery is enacted from

WfID - T_A N_1 - T_B N_2 -

then the Version for the next executing task will resemble

WfID - T_A N_1 - T_B N_2 - **Re** - T_C N_3 -

Because of the timestamps and node information, we have a guarantee that even if two distinct branches execute the same tasks in the same order, there can be no confusion about which Version(s) are valid for restarting each branch. If Monitors know the Version name of a failed task, they can obtain the results from a suitable previous execution without confusion. Finally, the Workflowid provides a way to differentiate executions of different workflows on the same WfMS.

The versioning scheme may also be helpful in identifying rogue or unwanted executions of the workflow. Such executions may result from trying to recover from network partitions or transport mechanism failure, for example. In such cases, nodes that are still active might look like they have failed to our monitoring mechanisms and as such multiple instances of the same execution may be executing at once. Imagine two tasks coming to the same node, with versions identical to a certain point (e.g., before the partition occurred). That node would notice a Version Clash, i.e., tasks share portions of their version where they should not have. Both executions have similar version roots, (WfID - $T_A N_1$ - $T_B N_2$ - $T_C N_3$ - $T_D N_4$ - …), but differ for work done after the partition. The recovery branch would have a "Re", while the baseline execution one (which did not really fail) would not. By checking for identical roots between two such branches, we can identify unwanted branches in the execution.